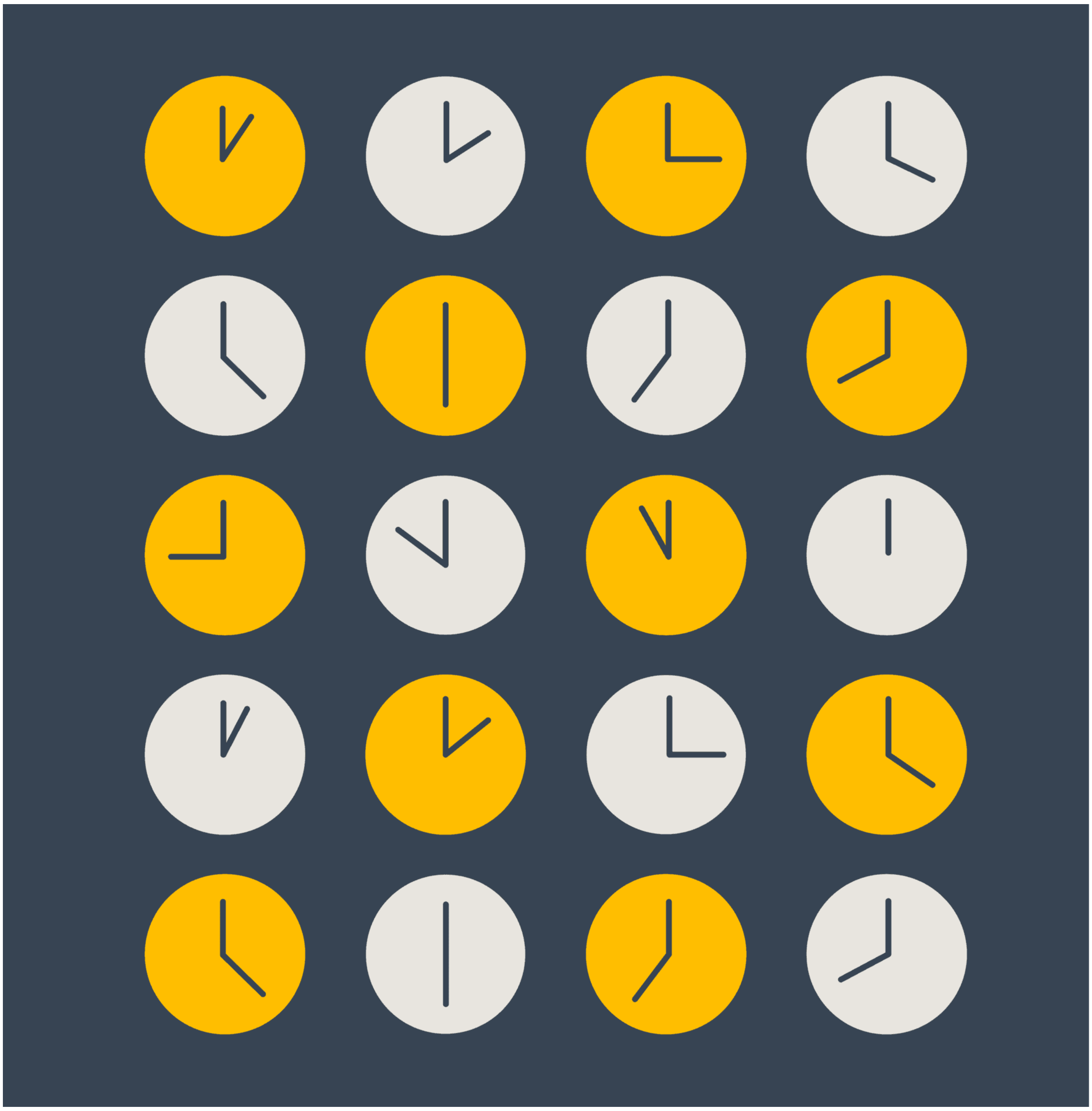
20 Things I've Learned in my 20 Years as a Software Engineer

Follow us

'



Important, Read This First

You’re about to read a blog post with a *lot* of advice. Learning from those who came before us is instrumental to success, but we often forget an important caveat. Almost all advice is contextual, yet it is rarely delivered with any context.

“You just need to charge more!” says the company who has been in business for 20 years and spent years charging “too little” to gain customers and become successful.

“You need to build everything as microservices!” says the company who built a quick monolith, gained thousands of customers, and then pivoted into microservices as they started running into scaling issues.

Without understanding the context, the advice is meaningless, or even worse, harmful. If those folks had followed their own advice early on, they themselves would likely have suffered from it. It is hard to escape this trap. We may be the culmination of our experiences, but we view them through the lens of the present.

So to give you a little context on where my advice comes from, I spent the first half of my career as a software engineer working for various small businesses and startups, then I went into consulting and worked in a number of really large businesses. Then I started Simple Thread and we grew from a team of 2 to a team of 25. 10 years ago we worked with mostly small/medium businesses, and now we work with a mix of big and small businesses.

My advice is from someone who...

1. has almost always been on small, lean teams where we have to do a lot with very little.
2. values working software over specific tools.
3. is starting new projects all the time, but also has to maintain a number of systems.
4. values engineer productivity over most other considerations

My experiences over the last 20 years have shaped how I view software, and have led me to some beliefs which I've tried to whittle down to a manageable list that I hope you find valuable.

On with the list

1. I still don't know very much

"How can you not know what BGP is?" "You've never heard of Rust?" Most of us have heard these kinds of statements, probably too often. The reason many of us love software is because we are lifelong learners, and in software no matter which direction you look, there are wide vistas of knowledge going off in every direction and expanding by the day. This means that you can spend decades in your career, and still have a huge knowledge gap compared to someone who has also spent decades in a seemingly similar role. The sooner you realize this, the sooner you can start to shed your imposter syndrome and instead delight in learning from and teaching others.

2. The hardest part of software is building the right thing

I know this is cliche at this point, but the reason most software engineers don't believe it is because they think it devalues their work. Personally I think that is nonsense. Instead it highlights the complexity and irrationality of the environments in which we have to work, which compounds our challenges. You can design the most technically impressive thing in the world, and then have nobody want to use it. Happens all the time. Designing software is mostly a listening activity, and we often have to be part software engineer, part psychic, and part anthropologist. Investing in this design process, whether through dedicated UX team members or by simply educating yourself, will deliver enormous dividends. Because how do you really calculate the cost of building the wrong software? It amounts to a lot more than just lost engineering time.

3. The best software engineers think like designers

Great software engineers think deeply about the user experience of their code. They might not think about it in those terms, but whether it is an external API, programmatic API, user interface, protocol, or any other interface; great engineers consider who will be using it, why it will be used, how it will be used, and what is important to those users. Keeping the user's needs in mind is really the heart of good user experience.

4. The best code is no code, or code you don't have to maintain

All I have to say is "coders gonna code." You ask someone in any profession how to solve a problem, and they are going to err on the side of what they are good at. It is just human nature. Most software engineers are always going to err on the side of writing code, especially when a non-technical solution isn't obvious. The same goes for code you don't have to maintain. Engineering teams are apt to want to reinvent the wheel, when lots of wheels already exist. This is a balancing act, there are lots of reasons to grow your own, but beware of toxic "Not Invented Here" syndrome.

5. Software is a means to an end

The primary job of any software engineer is delivering value. Very few software developers understand this, even fewer internalize it. Truly internalizing this leads to a different way of solving problems, and a different way of viewing your tools. If you really believe that software is subservient to the outcome, you'll be ready to really find “the right tool for the job” which might not be software at all.

6. Sometimes you have to stop sharpening the saw, and just start cutting shit

Some people tend to jump into problems and just start writing code. Other people tend to want to research and research and get caught in analysis paralysis. In those cases, set a deadline for yourself and just start exploring solutions. You’ll quickly learn more as you start solving the problem, and that will lead you to iterate into a better solution.

7. If you don’t have a good grasp of the universe of what’s possible, you can’t design a good system

This is something I struggle with a lot as my responsibilities take me further and further from the day to day of software engineering. Keeping up with the developer ecosystem is a huge amount of work, but it is critical to understand what is possible. If you don’t understand what is possible and what is available in a given ecosystem then you’ll find it impossible to design a reasonable solution to all but the most simple of problems. To summarize, be wary of people designing systems who haven’t written any code in a long time.

8. Every system eventually sucks, get over it

Bjarne Stroustrup has a quote that goes "There are only two kinds of languages: the ones people complain about and the ones nobody uses". This can be extended to large systems as well. There is no "right" architecture, you'll never pay down all of your technical debt, you'll never design the perfect interface, your tests will always be too slow. This isn't an excuse to never make things better, but instead a way to give you perspective. Worry less about elegance and perfection; instead strive for continuous improvement and creating a livable system that your team enjoys working in and sustainably delivers value.

9. Nobody asks "why" enough

Take any opportunity to question assumptions and approaches that are "the way things have always been done". Have a new team member coming on board? Pay attention to where they get confused and what questions they ask. Have a new feature request that doesn't make sense? Make sure you understand the goal and what is driving the desire for this functionality. If you don't get a clear answer, keep asking why until you understand.

10. We should be far more focused on avoiding 0.1x programmers than finding 10x programmers

The 10x programmer is a silly myth. The idea that someone can produce in 1 day what another competent, hard working, similarly experienced programmer can produce in 2 weeks is silly. I've seen programmers that sling 10x the amount of code, and then you have to fix it 10x the amount of times. The only way someone can be a 10x programmer is if you compare them to 0.1x programmers. Someone who wastes time, doesn't ask for feedback, doesn't test their code, doesn't consider edge cases, etc… We should be far more concerned with keeping 0.1x programmers off our teams than finding the mythical 10x programmer.

11. One of the biggest differences between a senior engineer and a junior engineer is that they've formed opinions about the way things should be

Nothing worries me more than a senior engineer that has no opinion of their tools or how to approach building software. I'd rather someone give me opinions that I violently disagree with than for them to have no opinions at all. If you are using your tools, and you don't love or hate them in a myriad of ways, you need to experience more. You need to explore other languages, libraries, and paradigms. There are few ways of leveling up your skills faster than actively seeking out how others accomplish tasks with different tools and techniques than you do.

12. People don't really want innovation

People talk about innovation a whole lot, but what they are usually looking for is cheap wins and novelty. If you truly innovate, and change the way that people have to do things, expect mostly negative feedback. If you believe in what you're doing, and know it will really improve things, then brace yourself for a long battle.

13. Your data is the most important part of your system

I've seen a lot of systems where hope was the primary mechanism of data integrity. In systems like this, anything that happens off the golden path creates partial or dirty data. Dealing with this data in the future can become a nightmare. Just remember, your data will likely long outlive your codebase. Spend energy keeping it orderly and clean, it'll pay off well in the long run.

14. Look for technological sharks

Old technologies that have stuck around are sharks, not dinosaurs. They solve problems so well that they have survived the rapid changes that occur constantly in the technology world. Don't bet against these technologies, and replace them only if you have a very good reason. These tools won't be flashy, and they won't be exciting, but they will get the job done without a lot of sleepless nights.

15. Don't mistake humility for ignorance

There are a lot of software engineers out there who won't express opinions unless asked. Never assume that just because someone isn't throwing their opinions in your face that they don't have anything to add. Sometimes the noisiest people are the ones we want to listen to the least. Talk to the people around you, seek their feedback and advice. You'll be glad you did.

16. Software engineers should write regularly

Software engineers should regularly blog, journal, write documentation and in general do anything that requires them to keep their written communication skills sharp. Writing helps you think about your problems, and helps you communicate those more effectively with your team and your future self. Good written communication is one of the most important skills for any software engineer to master.

17. Keep your processes as lean as possible

Everyone wants to be agile these days, but being "agile" is about building things in small chunks, learning, and then iterating. If someone is trying to shoehorn much more into it than that, then they're probably selling something. It isn't to say that people don't need accountability or help to work this way, but how many times have you heard someone from your favorite tech company or large open source project brag about how great their Scrum process is? Stay lean on process until you know you need more.

Trust your team and they will deliver.

18. Software engineers, like all humans, need to feel ownership

If you divorce someone from the output of their work, they will care less about their work. I see this almost as a tautology. This is the primary reason why cross-functional teams work so well, and why DevOps has become so popular. It isn't all about handoffs and inefficiencies, it is about owning the whole process from start to finish, and being directly responsible for delivering value. Give a group of passionate people complete ownership over designing, building, and delivering a piece of software (or anything really) and amazing things will happen.

19. Interviews are almost worthless for telling how good of a team member someone will be

Interviews are far better spent trying to understand who someone is, and how interested they are in a given field of expertise. Trying to suss out how good of a team member they will be is a fruitless endeavor. And believe me, how smart or knowledgable someone is is also not a good indicator that they will be a great team member. No one is going to tell you in an interview that they are going to be unreliable, abusive, pompous, or never show up to meetings on time. People might claim they have "signals" for these things… "if they ask about time off in the first interview then they are never going to be there!" But these are all bullshit. If you're using signals like these you're just guessing and turning away good candidates.

20. Always strive to build a smaller system

There are a lot of forces that will push you to build the bigger system up-front. Budget allocation, the inability to decide which features should be cut, the desire to deliver the "best version" of a system. All of these things push us very forcefully towards building too much. You should fight this. You learn so much as you're building a system that you will end up iterating into a much better system than you ever could have designed in the first place. This is surprisingly a hard sell to most people.

What is your story?

So there you have it, 20 years of software distilled down into 20 pithy pieces of wisdom. I'd love to hear it if something resonated with you. I'd also love to hear if you have a piece of wisdom that you've picked up over your career that you'd like to share. Feel free to leave it down in the comments.

Loved the article? Hated it? Didn't even read it?

We'd love to hear from you.

Reach Out